Harvey Mudd College
Computer Science 60
Section 1, Assignment 2

Due Friday, Feb. 8

Reading

Read through Chapter 4 in the text.

Getting started

This assignment contains a diverse set of problems (we have to keep up with Section 2, you know), so please get started early.

Practice Problems & Worksheet

If you would like a few more practice problems to try either before or after the assignment, there are several (with solutions) in `/cs/cs60/as/a2/practice.probs`. There is also a worksheet at

`http://www.cs.hmc.edu/courses/2002/spring/cs60/assignments/worksheets/w1.html`

For section 1, I am not requiring that you turn it in. However, you should know how to work all the problems in it just the same.

Testing your code

There is a test you can run `/cs/cs60/as/a2.1/test.rex` to try out your functions.

```
rex> assign2.rex < /cs/cs60/as/a2.1/test.rex
```

Test your code before submitting to increase confidence that your basic functionality is correct and to be sure there are no typos, misspellings, etc. and.

Submitting your code

You should submit your rex functions in a single file named `assign2.rex` using

```
cs60submit assign2.rex
```

Grading

As in assignment 1, commenting your functions and your file are important. Be sure your

comments indicate *what* each function does, as well as what the inputs are.

As you gain experience writing code in rex, in addition, it is a good idea (and very much the spirit of rex and functional programming) to try to devise programs as elegant and concise as possible.

Documenting your code

Your code should be documented according to the model described in

```
http://www.cs.hmc.edu/courses/2002/spring/cs60/examples/rex/acyclicTest.rex
```

Problems

1. Construct a function **`stackTwos`** so that `stackTwos(n)` is

   ```
   pow(2, pow(2, …., pow(2, 2)))
   ```

   for a total of n 2's. (pow(k, p) is k raised to the power p.) By definition, when n == 0 the result is 1. For n not terribly large, `stackTwos(n)` gets to be a pretty serious number. Find out for yourself the maximum `n` for which `stackTwos(n)` can be computed on turing.

2. Construct a function **`replicate`** that replicates a list n times, where n is the second argument. For example,

   ```
   replicate([1, 2, 3], 3)
         ==> [1, 2, 3, 1, 2, 3, 1, 2, 3]
   ```

   For n == 0, the result is, of course, the empty list.

3. Construct a function **`countOccurrences`** that counts the number of occurrences of an item in a list:

   ```
   countOccurrences("a", ["b", "a", "c", "a", "r", "a", "t"])
         ==> 3
   ```

4. Construct a function **`dropFirst`** such that `dropFirst(P, L)` drops only the first element in a list L satisfying predicate P:

   ```
   dropFirst(even, [1, 3, 5, 2, 4, 6]) ==> [1, 3, 5, 4, 6]
   ```

5. Construct a function **`maxFun`**`(F,G)` that takes two functions as input. Each function is a

function has a single numeric argument. `maxFun` returns *a function* that also has a single numeric argument, say `X`, and that returns the larger of `F(X)` and `G(X)`. For example here is some sample input and output:

```
rex>  foo(X) = 2*X + 1;
foo

rex>  bar(X) = 3*X - 2;
bar

rex>  baz = maxFun(foo, bar);
1

rex>  map(baz, range(1, 5));
[3, 5, 7, 10, 13]
```

6. A bowling game consists of a number of frames (10 to 12, depending on considerations to be explained). The objective of a given frame is to knock down as many of the 10 pins as possible, using at most two rolls. Ideally, all pins are knocked down on the first roll, and this is called a "strike". If not all pins are knocked down on the first role, then a second roll is allowed to try to get the remainder. If all pins are knocked down in two rolls it is called a "spare".

   Suppose we wish to represent the data for a bowling game by a list. Each element of the list represents one frame, and itself will be a list of one or two values, the number of pins knocked down on the corresponding roll. For example, we might have a game shown by the list:

   ```
   [[10],   [7, 2], [4, 0], [2, 7], [7, 3],
    [4, 5], [3, 7], [10],   [8, 2], [10],   [8, 0]]
   ```

   Here we have a strike in the first frame (beginner's luck, no doubt), a spare in the fifth frame, and so on.

   The hardest part of bowling is keeping score, so we enlist rex for this purpose. The score of a game is the sum of the scores of the frames. If a frame is neither a strike nor a spare, then the score of the frame is the number of pins knocked down. But if the frame contains a spare, the score is that number plus the number of pins knocked down on the next roll. Finally, if a frame contains a strike, the score is that number plus the number of pins knocked down on the next *two* rolls. So the maximum value of a frame is 30, in the case of a strike followed by two more strikes.

When the tenth frame is a spare, the player gets one additional roll to determine the score of the tenth frame. (The normal value of that roll is not added as if the roll were in a frame.) This shows up as one extra frame.

When the tenth frame is a strike, the player gets two additional rolls to determine the score of the tenth frame. This shows up as one or two extra frames, depending on whether the first roll is another strike.

Example: The list above would have a score of 142. [8, 0] is an extra frame due the strike in frame 10. The scores for each frame are as follows:

Frame Number	Score	Calculation
1	19	$10_{strike} + 7 + 2$
2	9	$7 + 2$
3	4	$4 + 0$
4	9	$2 + 7$
5	14	$10_{spare} + 4$
6	9	$4 + 5$
7	20	$10_{spare} + 10$
8	20	$10_{strike} + 8 + 2$
9	20	$10_{spare} + 10$
10	18	$10_{strike} + 8 + 0$

Construct the rex function **`bowlScore`** that takes a list of such frames and returns the overall score. Assume that the list is well-formed, i.e. that there are no extra rolls or deficient rolls, the number of pins is within range, etc.

In your definition, try to use pattern matching along the lines of the explanation. Your rules should serve as a formal explanation of how the game is scored. Don't worry about economizing on the number of rules, but do let recursion do the work for you. About seven or eight rules is reasonable.

7. In the game of scrabble, each word played receives a score which is the sum of the values

of its letters. The following table contains a list of lists of the form

`[ letter, this letter's value, number of this letter available ].`

For example, the letter 'b' is worth three points and there are two of them available in the game. (In this problem, you won't need to use the third component, the number of the letter available. You can use `assoc` on this list; it works with elements having one or more elements; not just two elements.)

```
letterValues =
    [
    ['a',1,9], ['b',3,2],  ['c',3,2], ['d',2,4], ['e',1,12],
    ['f',4,2], ['g',2,3],  ['h',4,2], ['i',1,9], ['j',8,1],
    ['k',5,1], ['l',1,4],  ['m',3,2], ['n',1,6], ['o',1,8],
    ['p',3,2], ['q',10,1], ['r',1,6], ['s',1,4], ['t',1,6],
    ['u',1,4], ['v',4,2],  ['w',4,2], ['x',8,1], ['y',4,2],
    ['z',10,1]]
```

Construct a function **`bestWord`** that will compute the highest-scoring word from a given list of letters (reperesented as a string), a list of useable words, and a list of letter values. For example,

```
bestWord("jbrquie", ["rue", "brie", "jibe", "ebi"], letterValues)
     ==> [13, "jibe"]
```

Note: The rex built-in function `explode` will expand a string into the list of letters contained in the string.

Developing the following two functions is suggested as an approach to solving the problem of producing `bestWord`. If you use them and use these names, we have test cases that can help you.

Function `scrabbleScore` computes the score for a given word relative to such a list. For example,

```
scrabbleScore("apple", letterValues) ==>  7
```

Function `findMaxWord` computes the highest scoring word from a given list of words relative to such a list. For example,

```
findMaxWord(["rue", "brie", "jibe", "ebi"], letterValues)
   ==>  [13, "jibe"]
```

The Unicalc Application, Part I (Part II will be on the next assignment)

The last three problems are related to the "Unicalc" unit-conversion application.

The Unicalc application converts quantities expressed in one type of unit to those of another. The end result (in next week's assignment) will be able to do conversions such as the following:

```
rex> convert( [1, ["mile"], ["hour"]], [1, ["meter"], ["second"]], db);

multiply by 0.447032 or divide by 2.23698
```

Unicalc works on multiplicative conversions only, so it does *not* perform additive conversions such as Fahrenheit to Celsius.

Functional programming provides a very natural way to implement the kernel of Unicalc. In this assignment, we walk through some of the data representations and issues, and ask you to write rudimentary code in rex. Here are the key representations used in Unicalc:

- A **Quantity List** is a list of three elements representing some quantity of a designated unit. For example,

```
[1.0, ["mile"], ["hour"] ]                 represents 1 mile per hour

[2.5, ["meter"], ["second", "second"] ]    represents 1 meter per second^2

[60.0, [ ], ["second"] ]                   represents 60 hertz

[3.14, [ ], [ ] ]                          represents about pi radians
```

To summarize, a Quantity List is a list of three elements: the first is the quantity (floating point), the second is a list of units in the numerator, and the third is the list of units in the denominator. **Note:** You can assume that the lists of units (within the numerator and within the denominator) are initially in alphabetical order. Your code should make sure this property is maintained, for future purposes

- A **Conversion Database** is a list of relationships between units. For example,

```
db =
[
  ["foot",    [12.0,       ["inch"],   []]],
  ["mile",    [5280.0,     ["foot"],   []]],
  ["coulomb", [1.0,        ["ampere", "second"],     []]],
];
```

Each line in the database is a list of two elements -- the first element is a string that names some unit and the second element is a Quantity List that is equivalent to that named unit.

Not every unit is defined in terms of others in the database. Some are intentionally left undefined. We'll call the latter **basic** units. In the above example, `inch`, `ampere`, and `second` are basic units. A database does not need to contain conversions for everything to everything; that would tend to make it very large and maintenance would be error-prone. Instead, all conversions done by Unicalc are derived from basic ones in the database.

8. The representation of a quantity is said to be **simplified** if there is no element common to its numerator and denominator lists. One way to convert a quantity to an equivalent simplified form is to "cancel" any pairs of elements common to its numerator and denominator. Construct a rex definition for a function **`simplify`** of one argument (i.e., one quantity list) that yields a corresponding simplified representation. I would suggest adapting the built-in function `merge` in order to take advantage of the fact that the numerator and denominator lists are sorted.

   Some example uses of `simplify` include

```
rex>   simplify([3, ["kg", "meter", "second", "second"],
                    ["kg", "kg", "second"]]);
[3, ["meter", "second"], ["kg"]]

rex>   simplify([3.14, ["meter", "second"],
                       ["meter", "second", "second"]]);
[3.14, [], ["second"]]
```

9. Construct rex functions **`multiply(QL1, QL2)`** and **`divide(QL1, QL2)`** that respectively multiply and divide two quantity lists, yielding a simplified result. Again, you may increase your code's efficiency by using `merge` to combine two sorted lists into a single sorted list. However, as with `simplify`, any technique is OK. Also, you may assume that the first element of a quantity list will never be 0.

   Some example runs include

```
rex>   multiply([2.0, ["kg"],    ["second"]],
                [3.0, ["meter"], ["second"]]);
[6.0, ["kg", "meter"], ["second", "second"]]

rex>   divide([12.5, ["meter"], ["second"]],
              [0.5,  ["meter"], []]);
[25.0, [], ["second"]]
```

   (Hint: don't rewrite `simplify`!)

10. As mentioned, the Unicalc database is represented as an association list, that is, a list of pairs in which each pair consists of a single unit and an equivalent quantity list for that unit. Essentially, these pairs define equations for various units, and we can refer to the elements of the pairs as the LHS (left-hand side) and RHS (right-hand side) for this reason. For example, a sample database might be defined as:

```
db =
  [
  ["foot",    [12.,       ["inch"],                 []]],
  ["mile",    [5280.,     ["foot"],                 []]],
  ["inch",    [0.0253995, ["meter"],                []]],
  ["hour",    [60.,       ["minute"],               []]],
  ["minute",  [60.,       ["second"],               []]],
  ["mph",     [1.,        ["mile"],                 ["hour"]]],
  ["coulomb", [1,         ["ampere", "second"],     []]],
  ["joule",   [1,         ["kg", "meter", "meter"], ["second", "second"]]],
  ["ohm",     [1,         ["volt"],                 ["ampere"]]],
  ["volt",    [1,         ["joule"],                ["coulomb"]]],
  ["watt",    [1,         ["joule"],                ["second"]]]
  ];
```

which happens to be the current contents of `/cs/cs60/as/a2/unicalc_mini_db.rex`. There is a substantial database of elements that you are also welcome to try out in the file `/cs/cs60/as/a2/unicalc_db.rex` .

If this were the entire database under consideration, note that `"ampere"`, `"meter"`, and `"second"` are all basic quantities, since they are not the LHS of any definition.

Construct a rex function **`conv_unit(u, DB)`** that takes a string representing a unit and the database association list as arguments. If the unit is defined in the database, it returns that unit's equivalent quantity list, otherwise it returns a standard quantity list representing the unit: `[1.0, [u], []]`. For example, if db is the database consisting of the list of the pairs above, then:

```
rex>   conv_unit("hour", db);
[60, [minute], [] ]

rex>   conv_unit("meter", db);
[1, [meter], [] ]
```

Assignment 3 will use these functions in order to complete the unit-conversion application.